

PACHA : Low cost bare metal development for shared memory manycore accelerators

A. Aminot, A. Guerre, J. Peeters, Y. Lhuillier

▶ **To cite this version:**

A. Aminot, A. Guerre, J. Peeters, Y. Lhuillier. PACHA : Low cost bare metal development for shared memory manycore accelerators. Procedia Computer Science, 2013, 18, pp.1644-1653. 10.1016/j.procs.2013.05.332 . cea-01831560

HAL Id: cea-01831560
https://cea.hal.science/cea-01831560

Submitted on 6 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1644 – 1653

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

PACHA : Low Cost Bare Metal Development for Shared Memory Manycore Accelerators

Alexandre Aminot[a], Alexandre Guerre[a], Julien Peeters[a], Yves Lhuillier[a]

[a]*CEA, LIST, Embedded Computing Laboratory, F-91191 Gif-sur-Yvette, France.*

Abstract

Today, efficiently implementing an application on shared memory manycore accelerators is a hard task. Linux eases the development, but is not adapted to exploit the maximum of the platform yet. *Bare metal* programming environments provided with accelerators give a low-overhead access to the platform. However, they increase the complexity of development, mainly due to 4 gaps: each accelerator has its own specific environment; bare metal environments are only supported on the hardware platform or on the proprietary simulator; they have seldom, if ever, execution debugging support; they do not have parallel programming models, and whenever they exist, they are platform-specific.

In order to fill the gaps and thus, to lower the barrier to develop on bare metal on shared memory manycore accelerators, we present PACHA. It features two aspects: a low overhead, Platform Agnostic, Close-to-HArdware (PACHA) programming interface, which allows to handle only one version of the application code for all supported accelerators, and an easy-to-use multi-platform development environment, which abstracts the complexity of each accelerator's development environment. With a x86 support and a Linux compatibility, PACHA offers a functional simulator and all the Linux set of debugging tools. Further, based on the programming interface, three parallel execution models have been ported in order to facilitate the development and comparison of applications. PACHA is currently fully supported for x86 platforms, TILEPro64 and STHORM. A case study on a TILEPro64 is presented: the performance gain using PACHA rather than Linux with OpenMP or Pthread is about 1,8x to 4x, without increasing the development cost.

Keywords: Bare-Metal; Accelerators; Manycore; Functional simulator; Benchmarking; Multi-platform; Programming interface; Development environment

1. Introduction

Nowadays, embedded systems are exploited to execute high performance applications with strong power constraints. To meet this challenge, shared memory symmetric manycore accelerators are proposed, such as the STHORM platform created by STMicroelectronics and CEA, formerly known as Platform 2012 [1] , Tilera processors [2], Kalray processors [3], or Octeon III [4]. In order to efficiently exploit these platforms, it is required to program and use them correctly, which implies choosing the right triplet {*application, parallel execution model, accelerator*}. To find it, an accurate solution is to test the application with different parallel execution models on several accelerators. This task requires a long development time, often without fitted tools. Indeed, each accelerator has its particular programming and development environments.

*Corresponding author. Tel.: +3-316-908-0071 ; fax: +3-316-908-8395.
E-mail address: alexandre.aminot@cea.fr.

1877-0509 © 2013 The Authors. Published by Elsevier B.V. Open access under CC BY-NC-ND license.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.332

Although Linux is often supported by the accelerators in order to simplify programming, it is not yet adapted to efficiently implement an application on manycore due to its considerable overhead. Constructors often propose a close-to-hardware programming environment to exploid the accelerator efficiently. On Tilera, there is the "Bare Metal Environment" (BME) and On STHORM, there is a low-level runtime. They provide basic primitives to access hardware without an operating system (OS) but they increase the difficulty to program and compare the architectures.

Bare metal environments have four issues: (1) They provide similar concepts, but with different interfaces. For the programming interface, the names of data types and functions are different. This forces the developer to write a new version of the application and learn how to use a new programming environment. Handling several versions of an application implies complex updates. For the development environment, the configuration of the platform, the options and the way of compiling are also specific to each platform. (2) Bare metal environments are only supported by the hardware platform or by the proprietary simulator. So, if the project includes several developers, they have to share the chip or use a potentialy heavy simulator. Plus, running or debugging tasks on a slow simulator can be a huge waste of time. (3) They are proposed to exploit the platform without degrading the platform performances, so most of them do not have native execution debugging and profiling support. (4) Bare metal environment do not have parallel programming models, and whenever such models are proposed, they are platform specific. Thus, bare metal environments require more efforts and are time consuming to develop an application.

In order to reduce the development cost and ease the benchmarking of an application on shared memory manycore accelerators, we propose PACHA, which contains a near zero overhead Platform Agnostic Close-to-HArdware (PACHA) programming interface, an easy-to-use multi-platform development environment and lightweight parallel execution models. PACHA is developed in the C language. A back-end for each platform makes connection between PACHA's Interfaces and the platform. PACHA makes the following contributions:

- *Opaque unified access to bare metal environment*: The PACHA programming interface abstracts the different usages and the complexities of each platform bare metal interface without extra overhead. Only one version of the application code is handled for all supported accelerators.
- *Functional simulator*: The user is not dependent on the hardware platform or the proprietary simulator. The user can run the application on the host machine (typically Linux workstation).
- *Complete set of debugging tools*: The user also can run the application using the Linux support of the target platform or on the host machine to take advantage of the debug tools provided by Linux.
- *Unified parallel programming models*: Based on the platform-agnostic programming interface, such models ease the comparison and the development of an application on several platforms.

Developping on Bare metal and benchmarking application on multiple platforms become available at a low-cost with PACHA. The interface is minimalist, integrating a new platform is thus fast, approximately 600 lines of code. Currently, four back-ends are implemented in PACHA: TILEPro64 Linux, TILEPro64 Close-to-hardware, STHORM simulator and x86 platforms.

The rest of this paper is organized as follows : Section 2 reviews the state of art in programming and comparing memory shared manycore accelerators. Section 3 details the PACHA programing interface and Section 4 presents the multi-platform development environment. Section 5 then illustrates PACHA with a case study on a TILEPro64. It presents an evaluation of the Linux and close-to-hardware back-ends and then compares the close-to-hardware back-end with programming parallel models to the currently most used way, using only Linux with Pthread or OpenMP. Section concludes and introduces our future work.

2. Programming and comparing shared memory manycore accelerators

Shared memory accelerators appear increasingly these last ten years, but only few fitted tools has been developed to help to correctly program and compare them. To compare general purpose manycores, there are benchmarks as Splash-2 [5], ParMibench [6] and Parsec [7]. They offer sets of heterogeneous applications, where a

comparison can be approximately done between the application to test and a similar application of the suite. However, they do not allow testing arbitrary applications and they are mainly based on Linux. MultiBench [8] targets the evaluation of scalable symmetrical multicore processors with shared memory. It is based on a multi-platform API [9] designed to run natively on any processor without an underlying operating system. MultiBench is limited to a thread-based programming model and gives only the opportunity to customize the included computation workload, it does not permit to develop a personal application.

To test a given application on several accelerators without handling many versions of the application's code, a minimum of abstraction is required. Linux (with Pthread or OpenMP) and virtual machines [10, 11] still have significant overhead and require specific support. There are multi-platform API initiatives aiming efficiency on manycore accelerators, such as OpenACC [12] which is based on code directives, MCAPI [13] which implies programming with message-passing and OpenCL [14] which aims to support heterogeneous platforms using an intermediate representation and runtime compilation.

Another way to be close to hardware and test an application on multiple accelerators is to port existing low-level runtime such as ExoKernel [15], Corey [16] or McRT [17]. However, it may take a lot of time because the software architecture may not be fitted, particularly the separation between functional and hardware-dependant parts. The most adaquate is McRT, which is already willing to run on multiple architecture.

Bare metal environments provided by chip vendors are a support for running applications without paying too much system overhead, so debugging and profiling supports are not native. Few tools exist such as [18] for NetLogic XLP and [19] for asymmetric multiprocessing architectures, using a core with Linux and shared information with baremetal cores. Another solution is to use functional simulators such as QEMU [20], but still complex manycores are hard to model.

PACHA brings a low-cost way to develop an application on baremetal, with only one version of the code running on several platforms. With a compatibility with Linux and a x86 support, a complete set of debugging tools and a functional simulator become available. The next section presents the main feature of PACHA, its platorm-agnostic close-to-hardware programming interface.

3. PACHA's programming interface

To exploit the platform without degrading performances, most constructors propose a close-to-hardware programming environment with similar concepts. They propose the minimum functions set to develop an application: access to memory, lock, information about the platform, traces, etc. However, these programming environments are provided by different interfaces and usages. To abstract these differences, we propose a thin Platform Agnostic Close-to-HArdware programming interface. It is presented in the first subsection. The features of each architecture are hidden and exploited thanks to the back-end of programming interface. Nevertheless, to give the user more flexibility, hardware features can be exploited explicitly by adding an extension to the programming interface. This is presented in the second subsection with an instance of the STHORM platform.

3.1. The programming interface description

Each low-level environment proposed by constructors have its own programing interfaces. In order to obtain a platform agnostic programming interface, our approach is based on :

- *Generic data types*: The concepts provided by the data is the same but data is not represented with exactly the same programming types, for example some use *uint16_t* to represent a lock, and others use a proprietary data type such as *bme_mutex_t* (TILEPro64). Generic data types allow different implementations of the interface without an overhead. They are implemented by each back-end, then at the compilation, the generic data types are replaced by the proprietary type. This can be seen as simple metaprogramming.

- *Generic set of functions*: Like data types, the available functions in specific bare metal environments provide the same concepts, but do not have the same names and number of arguments. A generic set of functions has to be defined. Using the generic data types allows to have a generic definition of functions. Note that using generic types forces the user to manipulate these generic data only with the functions provided by the programming interface. Moreover, several types of return values have to be explicitly defined with a

known type (i.e LibC type), which allows the user to print or use these return values in another parts of the code easily. Some return values are constant, such as error codes or status codes. These values are often platform dependent. For example the non-blocking function trying to acquire a lock, can return 0 when the lock is acquired (for STHORM) or 1 (for TILEPro64). So these values of constants have to become generic.

- *Generic execution context*: Independently of the application, an *init* (and *destroy*) function have to be implemented to hide all the action to do before the beginning of the real application. Plus, a generic entry point has to be defined to abstract the entry point of each accelerator. A generic entry point also allows a generic configuration of the platform. For example, if the platform is an emulated platform, the generic entry point hides the need of creating the simulated environment.

Table 1. Overview of the Pacha programming interface

Generic Types	Definition	Description
	lock	Represent a lock
	heapid	Represent an id of an heap
	coreset_t	Represent the set of core used for the application
	...	
	procid_t	Represent an id of core
Generic Functions		
Lock	lock_init(*lock**)	Init a lock
	...	
	lock_acquire(*lock**)	Blocking function, acquire a lock
	uint lock_tryacquire(*lock**)	Non blocking function, return LOCK_ACQUIRED or LOCK_BUSY
	LOCK_ACQUIRED	Macro, lock is acquired
Memory	*heapid* get_heapid(level)	Get the id of the heap lvl
	_SHARED(level)	Pre-process directive, indicate the memory location for a global variable
	...	
	malloc(heapid,size_t)	Allocate memory from the heap
	free(void * mem)	Deallocate the memory allocation pointed to by mem
Platform management	*procid_t* get_coreid()	Get the index of the caller core
	procid_t get_masterid()	Get the index of the master core
	...	
	uint NB_CORE_MAX	The maximum number of cores availaible for the application
Performance	uint64_t get_cycle()	Get the current cycle count
	uint32_t get_cycle_low()	Get the high 32 bits of the current cycle count
	uint32_t get_cycle_high()	Get the low 32 bits of the current cycle count
Task	exec_task(arg1,stack,task)	Execute a task with its proper stack, assembly implementation
Trace	pa_printf(const char *, ...)	Basic printf
	...	
	debug(uint lvl,const char *, ...)	Debug traces
	DEBUG_LVL_ACTIVED	Levels of debugging traces to print
Generic execution context		
	APP_ENTRYPOINT()	Entry point of the application, Pre-processor directive
	init()	Allow to init several parts of the environment
	destroy()	Clean the end of the execution

An overview of PACHA programming interface is presented Table 1. It is developed in the C language and defined by an explicit layer, which creates a clear separation between generic and hardware dependent-code. The explicit layer permits also to add easily generic code between the application and the back-end. This feature can be useful for debugging and profiling, for example, to profile the number of cycles spent to wait for a lock. The programming interface is based on the three principles presented above and allows to get only one version of the code for multiple accelerators without adding an overhead. The generic types are defined with *typedefs* . The pre-process replaces the generic type by the platform type, thus there is no overhead at the execution which allows linking with any data type: a proprietary type, a structure, a pointer or a scalar type. In Table 1, the generic types are italicized.

The generic set of functions contains the minimum set of functions in order to exploit the accelerator resources and to give the opportunity to create parallel execution models and complete application. The programming

interface is minimalist, Hence, integrating a new platform is fast. The average for our 4 implemented platforms is 600 lines of the C language and 15 of ASM. In order to implement a platform in PACHA, the minimum to have is a memory manager and a locks manager. If the platform doesn't have these libraries, approximately 500 lines of code must be added. The accelerator resources are reachable by libraries such as memory management, platform management, traces and performances. For synchronization and parallelism, libraries such as locks and tasks are available. Through this programming interface, no parallel execution model is fixed. It makes the assumption that the beginning of the code is executed by all the cores and then each core executes only the code written for it. Thus, multiple parallel execution models can be defined by software, such as pipeline chain (each core has a functionality and data flows through each processing stages), master-slave (a master creates and schedules tasks on different slave cores), fork-join, etc. The functions are defined by *static inline function* pointing toward the function implemented by the back-end. This layer does'nt cause an overhead because a call to a *static inline* function is replaced by its content (i.e the real call of the platform programming environment function). The constant values of the return of functions are defined by C pre-processor macros. For instance, *LOCK_ACQUIRED* has to be implemented by the back-end to define the value when a lock is acquired.

The generic execution context is created by *init* and *destroy* functions necessarily called at the start and end of the application. The generic entry point is defined by a C pre-processor directive, thus the correct entry point is well-interpreted by the proprietary compiler.

This presented API implies to implicitly use the special features of the platform. Indeed, a back-end uses the special features to implement the functionality of the API. Being generic does not permit to be always the most efficient, so special features would be used and tested explicitly. The next part details exploiting platform features by adding extensions.

3.2. Exploiting platforms features

This lightweight generic programming environment provides a minimalist interface. To allow a user to evaluate the application using a special feature of a given architecture, a special feature could be used through the programming interface or by adding an extension. Special features can be hide by the platform-agnostic programming interface, that is to say, they are employed to implement the interface. For example, in STHORM there is a hardware synchroniser (HWS) [21]. It is a hardware module dedicated to speed up generic synchronization constructs. It could be used for implementing the lock library. But, this hardware synchroniser brings more than the lock concept, it brings also events concept and hardware semaphores. Thus, another way to exploiting the specificities is to explicit specific features by adding an extension to the basic PACHA programming interface. It permits to test with or without the extension and put the concept to the test on other platforms. Indeed, for STHORM, an HWS extension has been added, which provides concepts such as hardware semaphores and events. For the platforms which do not have the feature, it can be emulated. For instance, on the TILEPro64 the event concept has been easily emulated with messages on the network. The emulation allows to handle only one version of code with the explicit feature.

The differences between accelerators do not only concern the programming environment, but also the development environment. To allow a non-expert developer to test the application on multiple accelerators, we propose an easy-to-use multi-platform development environment.

4. The development environment

Constructors provide with their platform a development environment containing their own software tools and methods. A development environment is mainly composed of tools to compile, configure the platform and start an application. They require a learning time before it could be used by a new user. Further, bare metal programming environments have a lack of debugging tools and parallel execution models. To reduce the development time, an easy-to-use multi-platform development environment is proposed. It consists in three concepts: (1) abstracting the proprietary development environment, (2) Linux compatibility and (3) platform independent parallel execution models.

4.1. Abstracting proprietary development environment

Development tools provided by chip suppliers can be different and require specific options. For example, for the configuration of the accelerators, TILEPro64 uses a specific descriptive file and STHORM uses compilation options. For the execution, each environment has its own method to transfer the code and start the accelerator: for a x86 platform, it's only an execution on the host. For TILEPro64, a launcher is involved. All these differences, require that developers go back to the documentation and adapt their development tools such as profiling scripts or makefile.

To abstract these differences, our multi-platform development environment has fixed a minimal set of actions. Thus, all the correspondences are done in the back-end implementation, and a user always employs the same actions regardless the targeted platform. Basic configurations are available to users by shared compilation options, such as the number of cores. For an expert exploitation of the platform, most of the options of each processor are available and custom configurations can be set up. For example, on TILEPro64, which contains a grid of 8x8 cores, it is possible to select any sub grid of the architecture. From the easy-to-use options, the TILEPro64 back-end generates automatically the proprietary description file to configure the platform.

This concept allows to switch from a platform to another with only one command. Plus, thanks to the platform agnostic programming interface, an application can be represented by one version of code. So, these two features offer a new way to reduce the development duration of an application: using Linux back-ends.

4.2. Linux compatibility to debug applications and speed-up development

Close-to-Hardware programming environments do not provide complete debugging tools, and are only supported by the hardware platform and its simulator. These two points can seriously slow down the development of an application. It is why a compatibility with Linux is proposed in our development environment. Indeed, PACHA permits to get only one version of code and quickly switch between platforms, thus if a problem does not depend on the back-end of the platform, it will be fixed thanks to the Linux back-end. Linux provides a complete suite of debugging tools and is widely available. To ease the porting of Linux back-end, a tool is added under the platform agnostic programming interface. Thanks to this tool, two back-ends have been ported in less than one week each, one back-end for x86 platform and one for TILEPro64 (over Linux). This tool deletes the abstraction realized by Linux that hides all cores behind its the *symmetric multiprocessing (SMP)* support, which is not the execution model provided by close-to-hardware environment. So, to be used as a debugging support, the tool creates emulated cores with the Pthread library. Each thread executes the application as if it were a real core. This method may not optimally exploit Linux, but it is especially useful for debugging.

4.3. Platform independent parallel execution models

The proposed programming interface does not impose any parallel execution model, but at least one is needed to develop an application on manycore architectures. Currently, three parallel execution models have been ported on the platform agnostic programming interface. They permit coarse and fine grain parallelism.

The first consists in classic, Posix-like, thread management functions for coarse grain parallel expressions. A global thread queue and core local queues are implemented. Each core tries to schedule a thread from its local queue, and if its local queue is empty, schedules a thread from the global queue. Each thread has its proper stack.

The second is the reactive tasks management (RTM) model [22]. It is a more appropriate model to exploit fine grain parallelism. It provides a fork-join framework, where each core schedules the tasks that the master has forked. When there are no more ready tasks, the master core waits until all tasks have finished their execution in order to enter the join task.

The third is ARTM [23], the asynchronous model of RTM. It implements asynchronous fork-join operations where the fork and join primitives are not necessarily executed by the same core. This characteristic allows to share cores between fork/join and so have a better resource usage.

Currently, these three parallel execution models are ported on the platform agnostic programming interface, which is implemented on four back-ends. The case study with a TILEPro64 presents the easy-to-reach gains using the models and software presented in this paper.

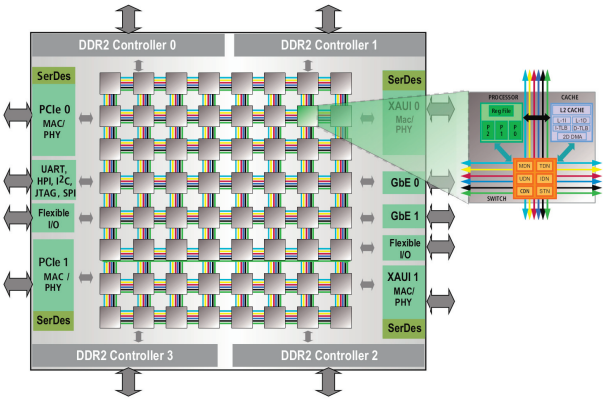


Fig. 1. Overview of the TILEPro64, created by Tilera. It incorporates a two-dimensional array of 8x8 cores, connected via multiple two-dimensional mesh networks. They are 6 independent networks : memory, cache coherence, user, instructions, inter-core and input/output. The platform has 4 DDR SDRAM controllers and a core has 3 levels of cache.Two physicals levels: an instruction L1 and a data L1, a mixed L2. One virtual L3, which is the set of all the other L2 cache of the others cores. The coherence is handled by a hardware coherence policy.

5. Case study on the Tilera TILEPro64

This section presents a case study of PACHA on the Tilera TILEPro64. An overview of the architecture is presented in Fig 1. We choose this platform for this case study because it is one of the first advanced shared memory manycore accelerator that supports Linux with the OpenMP and Pthread libraries. Two back-ends for the TILEPro64 have been implemented. The first supports the Tilera "Bare Metal Environment" (BME). It was implemented in two weeks, which includes learning time of BME. The second is a Linux back-end, using the Linux environment already ported by the Tilera team to the TILEPro64 architecture. It was implemented in few days, thanks to the Linux tools already present in the development environment (section 4.2).

We did not implement the application natively on BME because, as one of the problems that we have described before, BME has no debug tools and no parallel programming models. Writing the application on BME will require to rewrite a part of PACHA only dependent on the TILEPro64. The platform-agnostic programming interface has no overhead because the layer is replaced by calls to the platform environment at the pre-compile time. However, an overhead can be brought by the parallel programming models.

For this case study, we selected a pedestrian detection application, it is presented in the first part. With the help of this application, we evaluate the BME and Linux back-ends and then we compare the performance of the BME back-end to the classical parallel model methods on Linux, OpenMP and Pthread.

5.1. The pedestrian detection application

The pedestrian detection takes, as input, a VGA image (640×480 pixels) and gives as output boxes framing pedestrians in the image. The application is composed of a coarse grain part, the pre-processing, and a fine grain highly dynamic part, the classification. The pre-processing is composed of 21 coarse grain kernels in a pipeline chain. Each kernel implements an image processing algorithm like integral image calculation, Deriche filtering, etc. The aim of this part is to produce the 10 integral images required as input to the second part of the algorithm, the Viola-Jones classification cascade [24]. This part also takes as input Regions Of Interest (ROIs). For each one of the ROI, a 10 stages loop is executed in order to determine if this ROI contains a pedestrian or not. A positive box (one that actually contains a pedestrian) has to pass all 10 stages in order to be identified as such. Whereas a negative box can be rejected at any of the 10 stages. And since the processing complexity highly augments from one stage to the next, the classification part is hence dynamic in a sense that the processing power needed for each box is different and depends on the input data.

In our implementation, the classification is divided in 64 parts approximately equivalent in execution time. The sequential parts are minimal, thus the ideal speed-up is close to the number of dedicated cores. For the pre-processing, the 21 coarse grain kernels have many dependencies between them, the maximal speed-up is approximately 3.6.

5.2. Evaluation of the BME back-end and Linux back-end on a TILEPro64.

For these experimentations, the application is implemented with only one version of code. It is the back-ends that vary. The pre-processing part is implemented with the Pthread-like model and the classification part is implemented with the ARTM [23] model.

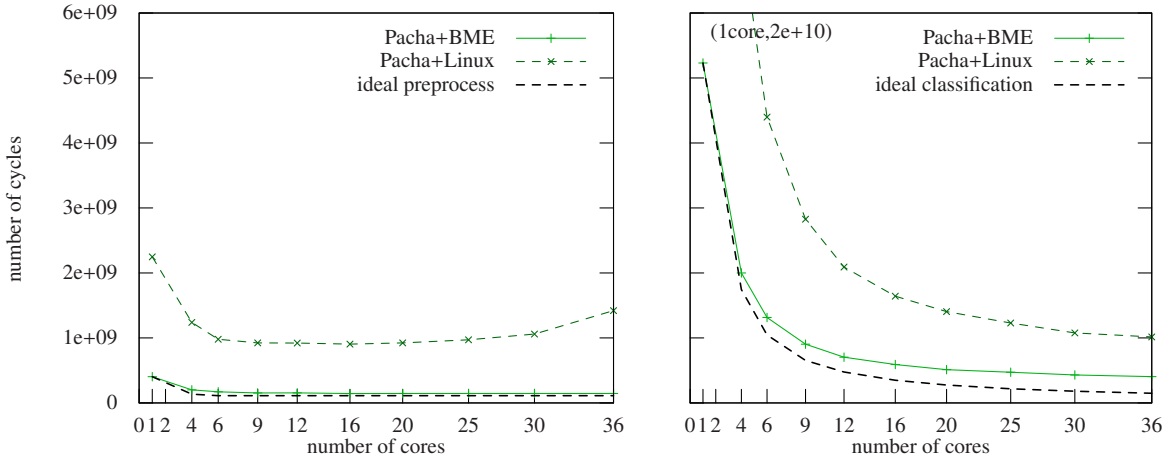


Fig. 2. Evaluation of the Linux back-end and the close-to-hardware back-end on TILEPro64. Execution time for (a) the pre-processing part (b) the classification part

Using the BME back-end, the maximum speed-up of the pre-processing is reached (Fig.2.a), but the speed-up is not maximal for the classification (Fig.2.b). A lightweight bare metal library has been created to profile the BME version, drawing inspiration from the OProfile [25] implementation for TILEPro64. An existing bare metal version of Oprofile developed by A. Schmidt exists [19] but it does not support the TILEPro64. Our library consists in using the hardware counters and it counts the apparition of an event (cache miss, stalls, packet send). We profiled each macro-block of the classification and calculated the median. On Fig.3, the profiling shows that the more cores are used, the more the execution time of a macro-bloc increases, which is due to data stalls. The cause could have been the cache coherence system [26], but the profiling of the networks used by the cache coherence system (cache and inter-core networks), Fig.3.b, shows only a slight increase of the packet send. However, the number of sent packet through the memory network increases similary to the number of cycles. So, the data stalls are due to the contention at the memory controllers. In fact, even if the data is distributed on the four controllers of the TILEPro64, and if each macro-bloc calculates an independent part of data, the number of accesses in parallel increases with the algorithm parallelization. So, the algorithm has to be rethought to run efficiently on the TILEPro64. With Linux Back-end on TILEPro64 or on a 8x8-core AMD Opteron (the speed up is 7x for 9 cores, 14x 20 cores, 19x 25 cores, 23x 56 cores), the speed up is more important because there is a huge overhead with the sequential version. For these versions, the contention appears starting from 25 cores.

Despite these contentions, the gain using the bare metal programming environment is important. With the Tilera BME back-end, the application is 2x to 4x faster than Linux (Fig.2). The implementation of the back-end with Linux may not be optimal, it is implemented especially to be close to the execution model of bare metal environment. So, we compare the implementation of the application with the BME back-end and the parallel programming models based on the common way of programming Linux: using OpenMP and Pthread libraries.

5.3. Comparison between the API in bare metal with common parallelism models

Three versions of the application were implemented over Linux, one OpenMP, one Pthread and one optimized Pthread. This last one employs the Tilera optimisation library to set the affinity of the thread on a core. These versions are independent of our solution, however the parallelism is operated the same way.

Fig.4 shows that the OpenMP and non-optimized Pthread versions are the less efficient. With more than 8 threads, there is no more gain and even worse, the execution time by macro-bloc gets bigger. The bare metal version is in a mojority of cases the best solution. Concerning the pre-processing, it is 4x faster. Concerning the Classification, the performance gain is about 1,1x to 2x. However, at 35 cores, the optimized Pthread version is as efficient as the close-to-hardware version. Nevertheless, the optimized Pthread version requires a higher cost of development than using PACHA. The BME version is developed without more optimization than a low-cost and naive development with the parallel programming models proposed by PACHA. With PACHA and a bare metal

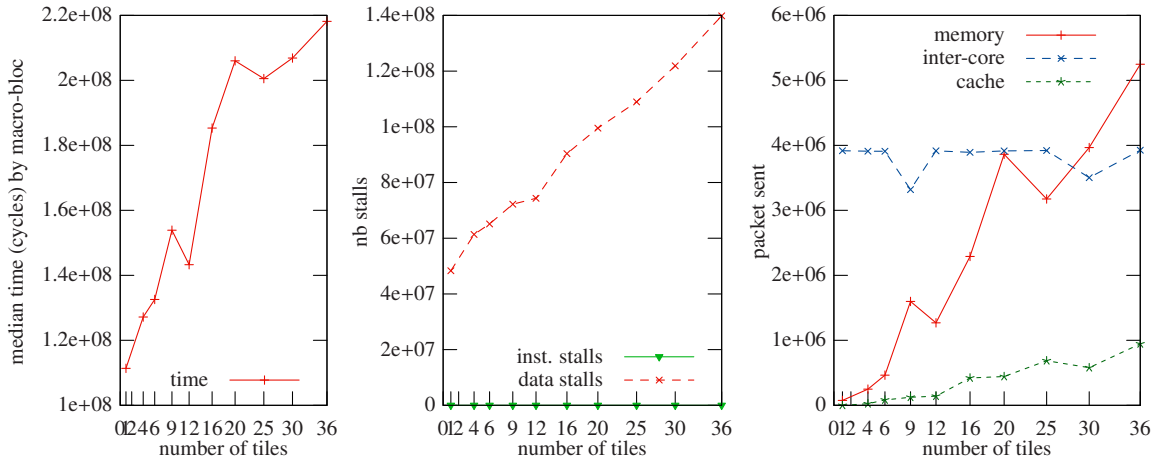


Fig. 3. Profiling macro-blocs of the classification, (a) the execution time median for a macro-bloc, (b) the instructions stalls and data stalls, (c) the network profiling, number of packets sent through cache coherency network, inter-core network and memory network

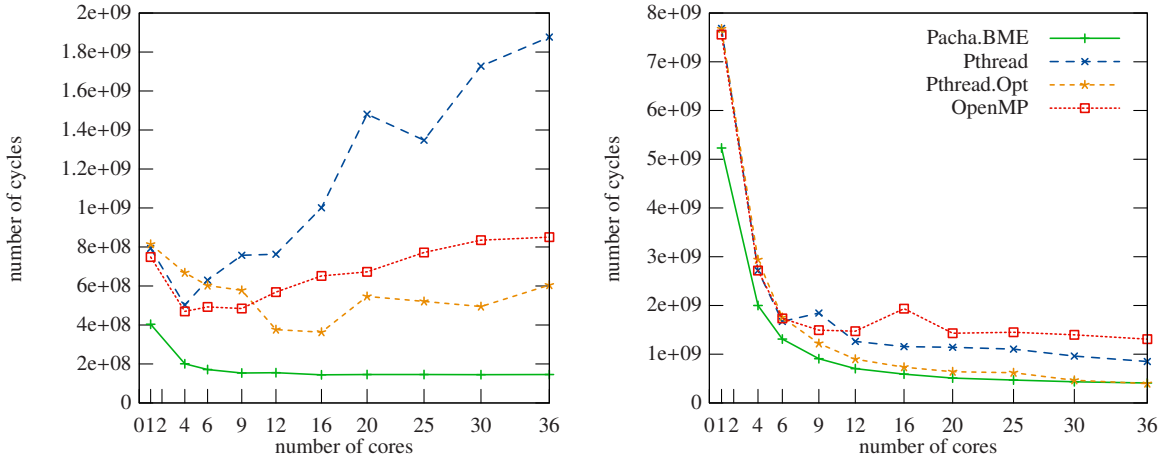


Fig. 4. Executing time of the pedestrian detection application running on a TILEPro64 with PACHA+BME or using Linux with common parallel programming libraries (a) Pre-processing (b) Classification

back-end, the performance is 1.8x to 4x better than non-optimized versions coded with Linux and OpenMP or Pthread.

6. Conclusion

Bare metal environments allow efficient access to dedicated hardware. To help developing an application on a bare metal environment, this paper presents PACHA. It contains a thin platform agnostic close-to-hardware programming interface and a multi-platform development environment. Benefits of PACHA for bare metal environment include unified access, debugging and profiling tools, a functional high speed simulator and unified parallel execution models. The development and the reuse of an application on bare metal environment and for multiple platforms become available at a low-cost. A case study on a TILEPro64 is presented with an application which detects pedestrians on a flux. Using PACHA with the bare metal environment rather than Linux with OpenMP or Pthread, the performance gain is about 1,8x to 4x.

In the future, we plan to continue our work focused on the development cost on bare metal environments and the benchmark of an application on multiple accelerators. To be more portable, PACHA is going to be C++

compatible and a SystemC support is being implemented to obtain another level of simulation, in order to perform energy profiling on applications.

References

[1] L. Benini, E. Flamand, D. Fuin, D. Melpignano, P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 983 –987.

[2] Tilera, 64-core processor, `http://www.tilera.com/`, [Online;accessed 10-dec-2012].

[3] Kalray, Multi-purpose processor array, `http://www.kalray.eu/en/products/mppa.html`, [Online;accessed 10-dec-2012].

[4] Cavium, Octeon multi-core processor family, `http://www.cavium.com/OCTEON_MIPS64.html`, [Online;accessed 10-dec-2012].

[5] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, SIGARCH Comput. Archit. News 23 (2) (1995) 24–36. doi:10.1145/225830.223990.

[6] S. Iqbal, Y. Liang, H. Grahn, Parmibench - an open-source benchmark for embedded multiprocessor systems, Computer Architecture Letters 9 (2) (2010) 45 –48. doi:10.1109/L-CA.2010.14.

[7] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: characterization and architectural implications, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, ACM, New York, NY, USA, 2008, pp. 72–81. doi:10.1145/1454115.1454128.

[8] T. R. Halfhill, Embc's multibench arrives, in: MPR Online, 2008.

[9] A. Weiss, The standardization of embedded benchmarking: pitfalls and opportunities, in: Computer Design, 1999. (ICCD '99) International Conference on, 1999, pp. 492 –508. doi:10.1109/ICCD.1999.808586.

[10] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis transformation, in: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, 2004, pp. 75 – 86. doi:10.1109/CGO.2004.1281665.

[11] D. Juhsz, llvm backend for tilera processor, `https://github.com/llvm-tilera`, [Online;accessed 10-dec-2012].

[12] CAPS Enterprise, Cray Inc., NVIDIA Corporation and the Portland Group, The openacc application programming interface, v1.0 (November 2011).

[13] M. Association, et al., Multicore communications api specification v1. 063 (mcapi) (2008).

[14] J. Stone, D. Gohara, G. Shi, Opencl: A parallel programming standard for heterogeneous computing systems, Computing in science & engineering 12 (3) (2010) 66.

[15] D. R. Engler, M. F. Kaashoek, J. O'Toole, Jr., Exokernel: an operating system architecture for application-level resource management, SIGOPS Oper. Syst. Rev. 29 (5) (1995) 251–266. doi:10.1145/224057.224076.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, et al., Corey: An operating system for many cores, in: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, 2008, pp. 43–57.

[17] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, J. Fang, Enabling scalability and performance in a large scale cmp environment, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 73–86. doi:10.1145/1272998.1273006.

[18] ENEA, Enea bare metal performance tools for netlogic xlp (2011).

[19] A. Schmidt, Profiling bare-metal cores in amp systems, in: System, Software, SoC and Silicon Debug Conference (S4D), 2012, pp. 1 –4.

[20] F. Bellard, Qemu, a fast and portable dynamic translator, USENIX, 2005.

[21] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe, R. David, An efficient and flexible hardware support for accelerating synchronization operations on the sthorm many-core architecture, to appear in Design, Automation Test in Europe Conference Exhibition (DATE), 2013 (march 2013).

[22] M. Ojail, R. David, K. Chehida, Y. Lhuillier, L. Benini, Synchronous reactive fine grain tasks management for homogeneous many-core architectures, ARCS 2011.

[23] M. Ojail, R. David, Y. Lhuillier, A. Guerre, Artm: A lightweight fork-join framework for many-core embedded systems, to appear in Design, Automation Test in Europe Conference Exhibition (DATE), 2013 (march 2013).

[24] P. Viola, M. Jones, Rapid object detection using a boosted cascade of simple features, in: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, Vol. 1, IEEE, 2001, pp. I–511.

[25] J. Levon, P. Elie, et al., Oprofile, A system-wide profiler for Linux systems. Homepage: http://oprofile.sourceforge.net [Online ; accessed 10-dec-2012].

[26] A. Agarwal, C. Celio, et al., Cache coherence strategies in a many-core processor, Ph.D. thesis, Massachusetts Institute of Technology (2009).